Implementación de MCTS y Redes Convolucionales para el desarrollo de una red neuronal en R

Amy Chen, Anthonny Flores, Leonardo Vega, Andrés Zúñiga
Escuela de Matemáticas
Universidad de Costa Rica
San José, Costa Rica

I. Introducción

El presente proyecto fue desarrollado con el objetivo de explorar los conceptos relacionados a las redes neuronales y los algoritmos de autoaprendizaje mediante la creación de un bot capaz de jugar partidas de ajedrez de manera competente. La idea surgió debido a que los miembros del grupo ya tenían conocimiento previo acerca de la complejidad del ajedrez y de lo avanzada que está el área de entrenamiento de bots para ese juego, además de que es bien sabido que este juego relaciona conceptos clave de la carrera de Ciencias Actuariales como lo son la estadística y la probabilidad de una manera que le resultaba interesante a los miembros del grupo. Adicionalmente, se consideró que el ir explorando el funcionamiento de las redes neuronales desde hoy en día es importante, pues según Awati (2025), la habilidad de adaptación con la que cuenta esta tecnología permite ser de utilidad al momento de analizar la información disponible del mercado de valores, reconocer patrones, gestionar el control de calidad de un proceso dado, hacer modelos financieros, combatir el fraude y buscar material para investigaciones científicas, entre otros. Todo lo anterior se relaciona a áreas en las que se desenvuelve un profesional en actuariado, y, por ende, es importante para el aprendizaje profesional de los miembros del grupo.

Una vez que se tuvo una idea inicial, se dio inicio a una investigación con el fin de entender de una mejor manera los conceptos asociados al ajedrez y redes neuronales, además de buscar paquetes de R que fueran de utilidad para la elaboración del proyecto, los cuales serán explicados a detalle en la sección de metodología y marco teórico del presente proyecto. Sumado a lo anterior, se hizo uso de herramientas externas como tarjetas gráficas con el fin de reducir el tiempo de entrenamiento del bot desarrollado y del motor de ajedrez Stockfish, este último orientado a probar las habilidades del bot.

Finalmente, se decidió implementar una interfaz de tal manera que el usuario pueda interactuar de manera directa con el bot mediante la ejecución de una partida, pues de esta forma se podía llegar a experimentar de primera mano el avance del proyecto, lo cual mejoraba el control de errores y proporcionaba un mayor dinamismo al trabajo en cuestión. Es por todo lo anterior que se le invita al lector a visualizar las siguientes páginas del presente proyecto, pues representa el esfuerzo y dedicación de los autores.

II. Metodología

Para el bot de ajedrez Hatchet1, el equipo se basó en el Aprendizaje por Refuerzo Profundo (Deep Reinforcement Learning), el cual toma inspiración del modelo AlphaZero para lograr un autoentrenamiento continuo (Self-Play). Este enfoque reemplaza el conocimiento heurístico humano programado con una Red Neuronal Profunda (NN) que se mejora iterativamente, guiando un potente algoritmo de Búsqueda de Árbol Monte Carlo (MCTS).

1. Red Neuronal Profunda y Codificación del Estado (NN)

El diseño de la Red Neuronal y la precisión de la Codificación del Estado son los pilares de la inteligencia de Hatchet1.

1.1. Codificación del Estado (FEN a Tensor) - Función fen.to.vector

La función fen.to.vector es esencial para la interfaz entre el motor de ajedrez (chess en R) y la NN. Transforma el estado FEN en un tensor $8 \times 8 \times 18$ (64 casillas, 18 canales), que es el formato de entrada esperado por keras3.

1.1.1. Canales de Piezas (1-12): Se utiliza una codificación one-hot para las 12 piezas. La asignación se realiza por la tabla map.piece, donde las piezas blancas se mapean a los canales 1 a 6 ("P"=1 a "K"=6) y las negras a los canales 7 a 12 ("p"=7 a "k"=12). El bucle recorre el FEN (rows) y asigna el valor 1 al canal correspondiente a la pieza y casilla.

1.1.2. Canales de Metadatos (13-18): Estos canales proporcionan información global que no se infiere de la distribución de piezas, siendo idénticos en todas las casillas (8×8).

- Canal 13 (Turno): Valor 1 si es el turno de las blancas ("w"), 0 si es el turno de las negras.
- Canal 14 (Enroque): Valor 1 si el enroque es legal para al menos un lado.
- Canal 15 (Repetición): Valor 1 si se detectan dos repeticiones de posición, anticipando la regla de tres repeticiones.
- Canal 16 (Constante): Una capa constante de 1, utilizada para ayudar a la red a detectar el límite del tablero.
- Canal 17 (Reloj de Medio Movimiento): El contador de movimientos sin captura o avance de peón (board.data[5]). Este valor se normaliza entre 0 y 1, utilizando un divisor de 100.0 ($\min(1, \text{clock}/100.0)$), crucial para que la NN aprenda la proximidad de la Regla de 50 Movimientos.
- Canal 18 (Número de Movimiento Completo): El contador total de movimientos (board.data[6]). Se normaliza con un divisor de 200.0, proporcionando una medida de la profundidad de la partida.

Finalmente, el array se reajusta (array_reshape) para coincidir con el tensor de lote requerido por keras3 para la predicción del modelo.

1.2. Arquitectura, Profundidad y Funciones de Activación

El cuerpo de la NN está compuesto por una Capa Convolucional Inicial y 20 Bloques Residuales (num.residual.blocks). La profundidad del cuerpo, con 20 bloques y 256 filtros, es crucial para la fuerza del modelo, similar a la arquitectura Hatchet1 de la competencia.

La elección de 20 Bloques Residuales permite a la red aprender representaciones jerárquicas complejas del tablero. Un Bloque Residual es fundamental en esta arquitectura para mitigar el problema del vanishing gradient al permitir que la información "salte" capas y facilitar el flujo del gradiente durante el entrenamiento.

Funciones de Activación ReLU: La función de activación principal utilizada en todas las capas internas del cuerpo y dentro de los bloques residuales es la Unidad Lineal Rectificada (ReLU). Esta función introduce no-linealidad de manera eficiente, ya que es simple y ayuda a mantener el entrenamiento estable.

$$\text{ReLU}(x) = \max(0, x)$$

Cabezales de Salida (Output Heads): Al final del cuerpo principal, la red se bifurca en dos cabezales de salida independientes, cada uno utilizando una función de activación final diferente:

- Cabezal de Política (Policy Head, p): Predice la probabilidad de cada movimiento legal. Utiliza la función Softmax para asegurar que la suma de las probabilidades de todas las jugadas legales sea 1.
- Cabezal de Valor (Value Head, V): Predice el resultado esperado de la posición para el jugador de turno, un valor entre -1 (derrota) y $+1$ (victoria). Utiliza la función Tangente Hiperbólica (Tanh):

  $$\text{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

  La función Tanh es seleccionada debido a que su rango de salida $[-1, 1]$ coincide directamente con la escala de los resultados del ajedrez, facilitando la cuantificación de la calidad posicional.

1.3. Optimización y Ponderación de Pérdida

La compilación del modelo utiliza el optimizador Adam con una tasa de aprendizaje de 0.0001. La Función de Pérdida es una combinación de mean_squared_error (para el valor V) y categorical_crossentropy (para la política p). Ambas pérdidas se ponderan con 1.0 (loss_weights = list(value = 1.0, policy = 1.0)), indicando que el modelo debe optimizar por igual la precisión de la calidad del movimiento y la predicción del resultado.

2. Búsqueda de Árbol Monte Carlo (MCTS)

El MCTS es el algoritmo de decisión central, implementado en la función run.mcts y dependiente de la

función best.move. Para el agente, el MCTS es el componente principal que toma la recomendación de la NN y realiza 20 simulaciones (o las que se le haya configurado en NUM.SIMULATIONS) para entregar la mejor jugada.

2.1. Constantes de Control Global

Dos constantes globales controlan el equilibrio de la búsqueda:

- $C.PUCT = 2.0$: El Coeficiente de Exploración. Este valor, definido globalmente, equilibra la exploración de caminos nuevos (debido a una alta probabilidad P de la NN) y la explotación de caminos ya conocidos que han dado buen resultado (alto valor Q).
- $NUM.SIMULATIONS = 20$: Número de simulaciones realizadas por MCTS por cada movimiento, gestionando la profundidad del pensamiento en relación con el tiempo disponible.

El árbol de búsqueda se almacena en el entorno global mcts.tree, actuando como una hash map para acceder rápidamente a las estadísticas de nodo (N, W, P) a través de la clave FEN. Esto permite un acceso rápido a cuánto se visita un estado (N), cuánto se gana desde ahí (W), y la probabilidad inicial (P) que le dio la red.

2.2. Fases de MCTS y Mecanismos de Robustez

La función run.mcts encapsula las cuatro fases de la simulación.

2.2.1. Selección con PUCT: La selección se guía por la función calculate.ucb, que usa la fórmula PUCT. Esta fórmula dirige el recorrido en el árbol:

$$\text{UCB} = Q + C_{PUCT} \cdot P \cdot \frac{\sqrt{N_{\text{sum}}}}{1 + N_{\text{current}}}$$

El valor Q representa la parte de Explotación, mientras que el segundo término, donde intervienen C_{PUCT} y P, es la parte de Exploración. Se elige siempre el movimiento que dé el UCB más alto. El valor Q es calculado como el ratio de valor/visitas (W/N) del nodo actual.

2.2.2. Expansión y Parche de Asignación Segura: Si se llega a un nodo nuevo, uno que no se ha visitado (es decir, $\sum N = 0$), se procede a la Expansión. Se obtiene la predicción de la NN para ese nuevo estado. En este punto, es vital el Parche de Asignación Segura configurado, ya que la red puede producir una política con una suma de probabilidades insignificante:

1. Se recorta la política P_{model} solo a los movimientos legales.
2. Se verifica que la suma de esas probabilidades no sea insignificante (es decir, mayor que safety.threshold, que es $1e-4$).
3. Si la suma es adecuada, se Normaliza ($P \leftarrow P/\sum P$).
4. Si la suma es muy baja, se asigna una Política UNIFORME ($P \leftarrow 1/\text{num\_moves}$) a todos los movimientos, lo que garantiza que el algoritmo siga explorando robustamente.

2.2.3. Retropropagación y Anulación de Recompensa: Una vez que se llega al final de la simulación, la Retropropagación actualiza las estadísticas de visita y valor. El valor de recompensa (Z) se propaga hacia atrás. Para el jugador anterior, el valor se invierte ($Z \leftarrow -Z$) en cada paso (for (node.data in rev(path))).

- Penalización de Empates: Para desalentar un juego pasivo, si se llega a un empate por tablas (is_stalemate, is_seventyfive_moves, o is_fivefold_repetition), el valor se anula a $Z = -0.5$ en lugar de 0. Esta pequeña penalización promueve la preferencia por la victoria.

2.3. Estrategia de Auto-Juego y Control de Muestreo - Función best.move

Esta función toma el resultado del MCTS y es crucial para el Self-Play (auto-entrenamiento).

- Ruido de Dirichlet: Si la partida está en las primeras 30 jugadas (move.count < 30), se inyecta Ruido de Dirichlet a la política en la raíz. Esto promueve una exploración más amplia al inicio de la partida.
- Temperatura de Muestreo (high.temp): Controla la exploración vs. explotación a lo largo del juego:
    1. Modo Exploración (Jugadas iniciales, move.count < 40): Se usa Temperatura $T = 1.0$. El movimiento se elige aleatoriamente (sample) basado en las probabilidades de visita (P_{MCTS}), asegurando la exploración de jugadas prometedoras.

2. Modo Explotación (Juego avanzado, move.count ≥ 40): Simplemente se toma la jugada con el máximo número de visitas.

- Controles de Seguridad: Si los valores de P_{MCTS} son anómalos (NA, 0, o si el which.max falla), el bot elige un movimiento completamente aleatorio (sample(moves.list, size = 1)).

3. Generación, Almacenamiento y Evaluación de Datos

El sistema de datos (datos_ajedrez.R) y las librerías (chess, processx) garantizan la experiencia de juego persistente y analizable. Este módulo es vital para el aprendizaje a largo plazo de Hatchet1.

3.1. Estructuras de Datos y Persistencia

Se utilizan las funciones df.game.data y df.heavy.game.data para registrar los datos. La función data.to.txt se encarga de la persistencia, guardando los archivos de juego en un directorio específico de almacenamiento con el argumento append = TRUE, asegurando el crecimiento continuo del conjunto de entrenamiento sin sobrescribir.

- Datos Pesados (games.heavy.data): Aquí se registra información crítica. Por ejemplo:
  - Historial FEN (Tabla_totales): Secuencia de todas las posiciones de la partida, crucial para recrear y entrenar a la NN.
  - Tiempo de Movimiento Medio (Tiempo_movimiento_medio): Se calcula usando la mediana de los tiempos de CPU dedicados al MCTS en cada jugada, indicando la carga computacional.
- Además, el ELO del agente y el color con el que jugó se registran también en esta estructura.

3.2. Medición del Rendimiento

El rendimiento se cuantifica mediante el sistema ELO, calculado por la función elo.function. El agente se evalúa frente a un oponente base con un ELO de 400.

Para la actualización del ELO, se usa la fórmula tradicional, pero con un factor de peso $K = 20$ (la constante de ajuste).

$$R'_{\text{Hatchet1}} = R_{\text{Hatchet1}} + 20 \cdot (\text{Score} - E_{\text{Esperada}})$$

Aquí, R'_{Hatchet1} es el nuevo ELO, R_{Hatchet1} es el ELO actual, Score es el resultado real (que puede ser 1 por victoria, 0.5 por empate, o 0 por derrota), y E_{Esperada} es la probabilidad de victoria esperada contra el oponente (que sale de una fórmula basada en la diferencia de ELO). La lógica del cálculo asegura que estos resultados se utilicen correctamente en la fórmula.

III. Resultados

Durante la fase de pruebas, el modelo Hatchet1 fue evaluado en el modo bot_vs_bot_interno, donde la inteligencia artificial jugó exclusivamente contra sí misma. En todas las ejecuciones las partidas finalizaron de forma correcta y se registraron únicamente tablas: no hubo victorias ni derrotas para ninguno de los bandos, solo empates por repetición de jugadas (lo más común), por ahogado o por la regla de las 50 jugadas. Este comportamiento confirma que el sistema es capaz de generar movimientos legales de manera sostenida y completar partidas válidas sin provocar errores del motor de ajedrez ni fallos de ejecución.

El número de jugadas por partida se concentró aproximadamente entre 65 y 90 movimientos, por lo que no se observaron ni partidas extremadamente cortas ni encuentros excesivamente largos. En términos prácticos, esto indica que Hatchet1 logra sostener una partida completa sin colapsar en pocas jugadas, pero tampoco entra en ciclos indefinidos que alarguen artificialmente la duración del juego. A nivel computacional, el tiempo promedio por movimiento se mantuvo alrededor de 4–5 segundos, lo cual se considera eficiente para el entorno experimental utilizado, dado que permite un juego fluido sin incurrir en tiempos de respuesta excesivos.

Sin embargo, es importante resaltar que no se realizó un análisis sistemático de las posiciones intermedias de las partidas. No se midieron métricas tácticas ni posicionales (como ventaja material, evaluación de motor externo o precisión por jugada), por lo que no es posible afirmar que las posiciones hayan sido equilibradas desde un punto de vista ajedrecístico, este análisis se puede hacer a nivel externo, donde lo que demuestra es que inicia bien, con jugadas de libro pero después entra muy rápido en posiciones muy estratégicas donde hay muchas piezas confrontándose. Lo único que puede concluirse con certeza es que el modelo evitó caer en derrotas rápidas y que las partidas concluyeron por mecanismos de tablas, y no por jaque mate directo.

En cuanto al estilo de juego, el comportamiento del bot fue bastante aleatorio, algo que era de esperar en la etapa actual del proyecto. La red neuronal utilizada corresponde a una arquitectura profunda de tipo residual inspirada en AlphaZero, pero todavía se encuentra en una fase temprana de desarrollo: no ha recibido un volumen suficiente de autoentrenamiento ni de entrenamiento supervisado como para internalizar patrones estratégicos sólidos. En consecuencia, los movimientos generados se asemejan más a una exploración casi aleatoria del espacio de jugadas que a un juego estratégico coherente. Esta aleatoriedad explica por qué las partidas tienden a terminar en tablas: el modelo explora diversas continuaciones sin una planificación clara de largo plazo, pero al mismo tiempo no comete, de forma sistemática, errores triviales que conduzcan a derrotas inmediatas.

Finalmente, aunque la arquitectura residual de Hatchet1 es teóricamente robusta y se basa en modelos consolidados en la literatura, la evidencia empírica disponible sigue siendo limitada por las pocas partidas que se jugaron. Solo se evaluó un modelo entrenado bajo un conjunto restringido de partidas insuficientes y sin un estudio de progresión del rendimiento a lo largo de múltiples ciclos de entrenamiento. Por ello, todavía no es posible afirmar que la arquitectura sea estratégicamente sólida ni que el bot posea un nivel competitivo alto; los resultados obtenidos muestran más bien que el sistema es estable, funcional y eficiente en tiempo de cómputo, pero aún lejos de comportarse como un motor de ajedrez fuerte frente a oponentes avanzados.

IV. Discusión

Los resultados obtenidos permiten matizar el cumplimiento de los objetivos planteados al inicio del proyecto. Por un lado, el sistema desarrollado logró integrar de forma funcional los componentes clave de un agente de ajedrez inspirado en AlphaZero: codificación del estado en tensores, red neuronal residual profunda, algoritmo MCTS y un módulo de generación y registro de datos. El hecho de que todas las partidas en modo bot_vs_bot_interno concluyeran de forma correcta, sin errores de ejecución y mediante mecanismos reglamentarios (tablas por repetición, ahogado o regla de las 50 jugadas), evidencia que la arquitectura propuesta es coherente a nivel técnico y que la interacción entre R, keras3 y el paquete chess es estable.

Sin embargo, cuando se analiza la calidad ajedrecística del juego, los resultados son más modestos. El patrón de partidas largas que terminan en tablas, junto con la ausencia de victorias o derrotas claras, sugiere que Hatchet1 se comporta como un agente que explora de manera amplia el espacio de jugadas legales, pero sin una planificación estratégica sólida. La red neuronal residual diseñada (20 bloques y 256 filtros) tiene capacidad teórica suficiente para capturar patrones de alto nivel, pero el volumen de autoentrenamiento disponible fue limitado y no se complementó con entrenamiento supervisado sobre partidas humanas o de motores consolidados. En este contexto, la combinación de una política todavía poco informativa con un MCTS de sólo 20 simulaciones por movimiento hace que el árbol de búsqueda no explote en profundidad las posiciones, lo que se traduce en un estilo de juego que se percibe cercano a la aleatoriedad controlada.

Este comportamiento también se explica por ciertas decisiones de diseño del MCTS. La inyección de ruido de Dirichlet en las primeras jugadas y el uso de temperatura alta en la fase inicial de la partida son mecanismos habituales para favorecer la exploración en esquemas de autoaprendizaje. No obstante, cuando el modelo subyacente no ha alcanzado un nivel de entrenamiento suficiente, estos mecanismos terminan amplificando la variabilidad de las jugadas sin que exista una "dirección estratégica" clara que oriente la exploración. De forma similar, la penalización suave de las tablas (recompensa $Z = -0.5$) busca desincentivar el juego pasivo, pero con pocas iteraciones de entrenamiento su efecto no alcanza a reflejarse en un cambio de estilo: el agente evita colapsos tácticos inmediatos, pero no desarrolla secuencias de jugadas que materialicen ventajas en victorias.

Otro aspecto relevante de los resultados es que la evaluación se limitó prácticamente al comportamiento en auto-juego. No se llevó a cabo un análisis sistemático de las posiciones intermedias utilizando un motor externo ni se midieron métricas como ventaja material promedio, precisión por jugada o tasa de errores graves. Tampoco se construyó una curva de progreso del ELO del agente a lo largo de múltiples ciclos de entrenamiento. En consecuencia, las conclusiones que pueden extraerse sobre la "fuerza real" de Hatchet1 son necesariamente cautelosas: se puede afirmar que el sistema es estable, que genera partidas legalmente correctas y que evita derrotas rápidas, pero no que posea aún un nivel competitivo frente a oponentes humanos fuertes o motores consolidados.

Desde el punto de vista computacional, los resultados ponen en evidencia una tensión entre la ambición del modelo y las restricciones de la plataforma utilizada. La arquitectura residual profunda y el uso de MCTS exigen grandes volúmenes de partidas y una capacidad de cómputo considerable. En este proyecto, la implementación en R demostró ser especialmente costosa para el manejo intensivo de árboles de decisión: tiempos del orden de varios segundos por jugada y la necesidad de revisar numerosos nodos por partida limitaron de forma práctica el número de episodios de entrenamiento posibles. Esto generó una brecha entre el diseño teórico del agente (cercano a esquemas de última generación) y lo que efectivamente se pudo entrenar y evaluar en un semestre con recursos restringidos.

Finalmente, a nivel formativo, la experiencia puede interpretarse como una introducción valiosa más que como la construcción de un motor de alto rendimiento. El proyecto permitió a los autores familiarizarse con conceptos fundamentales del Aprendizaje por Refuerzo Profundo aplicados al ajedrez —codificación de estados, separación política/valor, combinación de pérdidas, búsqueda guiada por políticas, persistencia de datos y cálculo de ELO—, al tiempo que visibilizó las limitaciones de R para este tipo de aplicaciones y la importancia de alinear los objetivos de desempeño con los recursos disponibles. En este sentido, aunque el bot no alcanzó un nivel competitivo elevado, sí evidenció el potencial de la arquitectura y dejó una base técnica y conceptual sobre la cual podrían desarrollarse, en trabajos futuros, modelos mejor entrenados, migraciones a lenguajes más eficientes y esquemas de evaluación más rigurosos frente a oponentes externos y métricas estandarizadas.

V. Conclusiones

La primera conclusión a la que se llegó durante el desarrollo del presente proyecto fue que todavía hace falta una gran cantidad de aprendizaje para lograr usar de manera correcta el concepto de redes neuronales. Una de las mayores dificultades se dio en que, cada vez que se quería hacer un avance había que estudiar una serie de conceptos, los cuales a su vez tenían raíces en otros temas aparte. En síntesis, este tema es sumamente amplio, y en el transcurso de un semestre resulta sumamente difícil aprender a manejar la teoría detrás de las redes neuronales.

Luego, la segunda conclusión que se pudo generar fue que el lenguaje de programación R es sumamente ineficiente al momento de manejar árboles de decisión, pues tan solo recorrer un árbol completo tardaba 5 segundos en promedio. Lo anterior, sumado a que es necesario revisar una gran cantidad de árboles por juego, extendió el proceso de entrenamiento en gran medida. Esto, sumado al equipo y presupuesto con el que los miembros del grupo contaban, complicó en gran medida el desarrollo del proyecto. Cabe aclarar que esto es así específicamente en R, pues durante la investigación se llegó a apreciar que otros lenguajes como Python son capaces de recorrer los mismos árboles con un poder computacional similar en menor tiempo.

Partiendo de la conclusión anterior, fue que se determinó que no era posible cumplir con los objetivos planteados al inicio del proyecto, más específicamente no se pudo determinar de manera certera la eficacia del bot entrenado. Esto se debió a que, para poder brindar un resultado concluyente, era necesario realizar alrededor de 100000 sesiones de entrenamiento, lo cual habría requerido de una inversión de tiempo y de energía significativa.

Sin embargo, e independientemente de lo mencionado en el párrafo anterior, la elaboración del presente proyecto sirvió como una introducción a los conceptos más básicos del área de estudio de redes neuronales. Sumado a esto, y pese a que el proceso de entrenamiento no alcanzó el punto deseado por los autores, sí se pudo comprobar el potencial de este tipo de entrenamiento, pues el bot logró alcanzar correctamente el nivel de un jugador de ajedrez promedio pese a todas las limitaciones computacionales con las que los autores contaban.

1. Limitaciones

La mayor limitación que se llegó a enfrentar durante la elaboración del proyecto fue el tiempo con el que se contaba, pues esto limitó el proceso de aprendizaje de la teoría asociada a las redes neuronales, lo que eventualmente provocó que el progreso se viera frenado. Relacionado a lo anterior surgió otra limitación, la cual fue que, debido a la ineficiencia de R en el manejo de árboles de decisión y a los recursos con los que los miembros del grupo contaban, llegó a hacer falta una gran cantidad de tiempo y energía para concluir el entrenamiento del bot, lo cual no era viable para efectos del proyecto.

2. Recomendaciones

A nivel técnico, los miembros del grupo recomiendan usar otros lenguajes de programación para la

elaboración de un bot que deba cumplir con funciones similares a las buscadas en este proyecto, o para cualquier trabajo que tenga que usar árboles de decisión como herramienta principal para cumplir con su objetivo. De no ser posible lo anterior, se recomienda contar con un equipo computacional con una potencia considerable, pues en este tipo de trabajos es necesario tener un procesador y una tarjeta gráfica lo suficientemente competentes.

A nivel práctico, se recomienda elaborar este tipo de proyectos en grupo pues es un trabajo sumamente pesado para una única persona. Sin embargo, también se recomienda a los grupos que se esfuercen por contar con un buen nivel de comunicación y con un nivel similar en el ámbito de programación, pues a lo largo del desarrollo de este proyecto se llegaron a dar inconvenientes debido a la diferencia de conocimientos y errores al momento de comunicar ideas, lo cual fue especialmente problemático al momento de la división de tareas.

Referencias

[1] Awati, R. (27 de octubre del 2025). What is a neural network?. https://www.techtarget.com/searchenterpriseai/definition/neural-network

[2] Klein, D. (2022). Neural networks for chess: The magic of deep and reinforcement learning revealed. arXiv. https://arxiv.org/abs/2209.01506

[3] Aggarwal, C. C. (2018). Neural networks and deep learning: A textbook. Springer.

[4] Bishop, C. M. (2006). Pattern recognition and machine learning. Springer Science+Business Media.

[5] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT Press.

[6] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science, 362(6419), 1140–1144. https://doi.org/10.1126/science.aar6404

[7] Alam, S. (2023). An investigation of the imputation techniques for missing data. ScienceDirect. Recuperado de https://www.sciencedirect.com/science/article/pii/S2772662223001819

[8] DeLeo, M., & Guven, E. (s.f.). Learning chess with language models and transformers. Whiting School of Engineering, Johns Hopkins University.

[9] R Core Team. (s.f.). CRAN: Manuals – The Comprehensive R Archive Network. Recuperado de https://cran.r-project.org/manuals.html

[10] Chang, M., & Liang, T. (2018). Recreating AlphaZero Chess Engine. Reporte de proyecto del curso CS230: Deep Learning, Stanford University.

[11] Pomares C., M. (s.f.). Entropy and Economics. Preprint. Recuperado de https://arxiv.org/abs/2407.00022

\end{document}